Multiple Dispatch using Compile-Time Metaprogramming

Seyed H. Haeri (Hossein)[1] and Paul Keir[2]

[1] Université catholique de Louvain, Belgium. email: hossein.haeri@ucl.ac.be
[2] University of the West of Scotland, UK. email: paul.keir@uws.ac.uk

Abstract. We solve the multiple dispatch problem for a *components-for-cases* encoding of algebraic datatypes in C++. For the multi-method, the programmer is required to specify the decentralised match statements using overloads of template functions. Then, they employ a one-liner preprocessor macro, which expands to the real dispatcher. The expanded one-liner performs iterative pointer introspection to accomplish late-binding using overload resolution. Even though we present our solution for ADTs, one can use the same technology for other types too.

1 Introduction

A function call is multiply dispatched when more than one argument of it is late-bound. Implementation of programming languages with formal semantics is fertile ground for such functions. For example, suppose that one is about to implement lazy evaluation. For sharing of heap bindings, one may choose amongst no sharing of Abramsky and Ong ($s_\perp$) [1], sharing of Launchbury (s_ℓ) [26], and complete sharing of (the flawed semantics [17, §10.2] of) Sinot (s_κ) [37]. Then, a possible implementation is due to the pattern matching

$$
\begin{array}{l}
eval : \Sigma \times H \times K \to V = \ldots \mid \\
\textsf{case}(s_\perp, \Gamma, \lambda x.e)\{\ldots\} \quad \mid \quad \textsf{case}(s_\ell, \Gamma, \lambda x.e)\{\ldots\} \quad \mid \quad \textsf{case}(s_\kappa, \Gamma, \lambda x.e)\{\ldots\} \\
\textsf{case}(s_\perp, \Gamma, e\ x)\ \{\ldots\} \quad \mid \quad \textsf{case}(s_\ell, \Gamma, e\ x)\ \{\ldots\} \quad \mid \quad \textsf{case}(s_\kappa, \Gamma, e\ x)\ \{\ldots\}
\end{array}
$$

where Σ, H, K, V are the sets of sharing policies, heaps, expression kinds, and values respectively. The first and the third arguments above need late-binding.

Another example is structural equality check for expressions. Both arguments below need late-binding: (The above check is the running example of this paper.)

$$
\begin{aligned}
eq(\_,\_) &= \perp && (1)\\
eq(Num(n_1), Num(n_2)) &= n_1 == n2 && (2)\\
eq(Add(l_1, r_1), Add(l_2, r_2)) &= eq(l_1, l_2) \wedge eq(r_1, r_2) && (3)\\
eq(Mul(l_1, r_1), Mul(l_2, r_2)) &= eq(l_1, l_2) \wedge eq(r_1, r_2) && (4)\\
eq(Sub(l_1, r_1), Sub(l_2, r_2)) &= eq(l_1, l_2) \wedge eq(r_1, r_2) && (5)\\
eq(Neg(t_1), Neg(t_2)) &= eq(t_1, t_2) && (6)\\
&\ldots
\end{aligned}
$$

Another typical use case for multiple dispatch is "object collision in video-gaming." The exact visuals used for demonstrating the collision is determined by the dynamic types of **all** objects involved rather than just one of them. Thus, the dynamic dispatch (or late-binding) needs to be based on multiple objects rather than just one – which would be the familiar dynamic dispatch of OOP.

Multiple dispatch is a classical problem in object-oriented programming. To solve the multiple dispatch problem in C++, we leverage built-in services of a C++ compiler: overload resolution and pattern matching for `template` parameters. We use those two compile-time services in combination with runtime pointer introspection to emulate pattern matching of functional programming. Our pattern matching is orchestrated by our machinery rather than the compiler.

We present our solution here for a recent encoding of Algebraic Datatypes (ADTs) in C++ [18]. (ADTs are not C++ built-in. Nor do they have have definite C++ encodings.) The encoding is paired with a single dispatch machinery. We found it natural to extend that machinery to a multiple dispatch one. Still, our solution is not exclusively for ADTs or that specific encoding.

The recent encoding of ADTs that inspired us comes as a part of a solution to the famous Expression Problem (EP) [11,36,44]. That is a C++ version for a technique that was formerly developed in Scala: Integration of a Decentralised Pattern Matching (IDPaM) [22]. Hereafter, we refer to that as the C++ IDPaM. We refer to its Scala predecessor as the Scala IDPaM. For statements applicable to both the C++ and Scala IDPaM, we simply use IDPaM.

Our solution is particularly easy to use, extend, and reuse (§ 2). The underlying technology is less trivial with its high usage of C++ metaprogramming. We explain that in great detail (§ 3) to make it accessible to the unfamiliar reader. In particular, we employ term rewriting (§ 3.3) back and forth when it comes to explaining our dispatch code (§ 3.4). In § 3.5, we discuss some advantages and disadvantages of our solution. Our experience with two test cases is reported in § 4. Related work comes in § 5 and we end in § 6.

Our success factor in orchestrating the pointer introspection is the capability of C++ for specifying type lists (say, using `tuple`) and their traversal – both at compile-time. When the cases of an ADT are stored in a type list, one can traverse them programmatically (for pointer introspection) by pattern matching on that type list. Our technology is available to any language that (like C++) can support that programmatic traversal and introspection.

This paper assumes modest C++ understanding. Due to space restrictions, we drop the `namespace` specifier for the Standard C++ entities. For example, when we write `tuple`, we actually mean `std::tuple`.

2 Usage Overview

The C++ IDPaM has three major role players: the case components, the ADT definition, and the match components. In presence of those three role players, a one-liner preprocessor macro is employed to instruct the late-binding for single dispatch. Our solution to multiple dispatch has the same three role players. Our

one-liner, however, is a generalisation to the one-liner of the C++ IDPaM. We now brief the reader to the role players, whilst our one-liner is detailed in § 3. The bits of C++ IDPaM that we use here and do not explain come in § A. One can refer to the original work [18] for even more on the C++ IDPaM.

Case Components IDPaM prescribes for the ADT cases to inherit from the ADT. Furthermore, IDPaM is inspired by Component-Based Software Engineering (CBSE) [40, §17],[35, §10]. Like previous EP solutions of Haeri and Schupp [19,17,21], IDPaM too takes a *components-for-cases* approach: that is, each ADT case is implemented using a standalone component (in its CBSE sense) that is ADT-parameterised, a.k.a., *case components*. In the C++ IDPaM, the ADT-parametrisation translates into type-parametrisation by ADT. For example, as the common EP showcase, here are the case components for numbers (`Num`) and addition (`Add`):

```
1  template<typename ADT = _> struct Num: ADT {//Num  α : ℤ → α
2    Num(int n): n_(n) {}
3    int n_;
4  };
5  template<typename ADT = _> struct Add: ADT {//Add  α : α × α → α
6    Add(const ADT& l, const ADT& r):
7      l_(msfd<ADT>(l)), r_(msfd<ADT>(r)) {}
8    const shared_ptr<ADT> l_, r_;
9  };
```

Notice how `Num` and `Add` both take `ADT` as a type parameter and inherit from it. This is a specific way of F-Bounding [8] commonly referred to in C++ as the Curiously Recurring Template Pattern [43, §21.2]. (See § A on `msfd`.)

Inheritance from the `ADT` is what legislates passing an instance of a case component for an instance of the `ADT`. This is required for composing expressions into more complex ones. As such, because of line 8 above, an instance of `Add` can store pointers to instances of **any** case of the `ADT` passed as the type parameter to `Add` in its `l_` and `r_` data members.

ADT Definition Suppose one is interested in having an ADT with Numbers and Addition as the only cases. Using the formalism of Haeri and Schupp [21], one describes that as: $NA = Num \oplus Add$. The necessary steps for implementing NA in C++ IDPaM follow:

```
10 struct NA: Namer {};
11 template<> struct adt_cases<NA>
12 { using type = tuple<Num<>, Add<>>; };
```

The above **`template`** specialisation of `adt_cases` for `NA` (line 12) is a meta-function instructing the compiler for `adt_cases` := `adt_cases` $\cup$ {`NA` $\mapsto$ $Num \oplus Add$}. That is, it introduces `tuple<Num<>, Add<>>` to the compiler as the *case list* of `NA`, enabling the compiler to infer the former from the latter, when required. Other case component combinations are done similarly, provided the presence of the additional case components. (See § A for `Namer`.)

Match Components For the implementation of functions defined on a datatype, IDPaM gets the programmer to instruct the late-binding. In the C++ IDPaM, this instruction is a one-liner that takes advantage of the built-in C++ overload resolution. That instructed late-binding is a one-off for each function defined on ADTs, no matter how many new case components are added later on. From a user's point of view, our one-liner for multiple dispatch has the same look and feel as that of the C++ IDPaM. For example, our one-off macro to expand for structural equality test is:

```
13 GENERATE_MULTIPLE_DISPATCH(bool, eq)
```

Under the hood, the one-liner assembles building blocks provided by the programmer. (C.f. § 3 for the macro details.) In short, the assembled building blocks form a structure akin to the familiar pattern matching of functional programming. The familiar pattern matching, however, is holistic; all the match statements are together in the pattern matching and the individual match statements do not exist elsewhere. One can essentially not detach the individual match statements from the holistic pattern matching. Selectively reusing the match statements is, hence, next to impossible. What we refer to as the building blocks are, on the contrary, independent of the resulting assembly they set up. Those are, again, components in the CBSE sense, a.k.a., the *match component*s.

In the C++ IDPaM, the match components are simply function overloads. So are also our match components. For example, here are the corresponding overloads of Equations 1–3:

```
14 template<typename C1, typename C2>                          //eq(_,_)
15 bool the_eq_match(const C1&, const C2&){return false;}     //= ⊥
16 template<typename ADT1, typename ADT2>      //eq(Num(n1), Num(n2))
17 bool the_eq_match(const Num<ADT1>& n1, const Num<ADT2>& n2)
18 {return n1.n_ == n2.n_;}                                  //= n1 == n2
19 template<typename ADT1, typename ADT2>//eq(Add(l1, r1), Add(l2, r2))
20 bool the_eq_match(const Add<ADT1>& a1, const Add<ADT2>& a2)
21 { return eq(*a1.l_, *a2.l_) &&                           //= eq(l1, l2) ∧
22          eq(*a1.r_, *a2.r_); }                             //eq(r1, r2)
```

That little code is enough for testing `NA` expressions for structural equality. For simplicity of programming, however, one can add syntactic sugaring. Armed with that, we get **`false`** for `eq(5_n, 2_n + 10_n)` and **`true`** for `eq(5_n, 5_n )`, as expected.

In summary, each match component corresponds to one and only one match statement – enabling *decentralisation* of a pattern matching. The set of match statements and their order, in IDPaM, is open to the programmer for configuration at the right time. Instead of it being delivered holistically, the pattern matching then is by *integration* of the (decentralised) match components. Hence, the IDPaM name.

2.1 Extensibility and Reuse

As mentioned earlier, the C++ IDPAM is an EP solution. The implication that is of particular interest here is that the existing codebase can be reused but not required to be recompiled upon the addition of new case components, match components, and ADT definitions. Suppose, for example, that case components are later added for Multiplication (`Mul`), Subtraction (`Sub`), and Negation (`Neg`). Suppose also that ADTs are defined for $NAM = NA \oplus Mul$, $NAG = NA \oplus Neg$, and $NAGS = NAG \oplus Sub$. Then, **all** that remains for the new ADTs to enjoy the structural equality check is the following overloads for Equations 4–6:

```
template<typename ADT1, typename ADT2>    //eq(Mul(l1,r1),Mul(l2,r2))
bool the_eq_match(const Mul<ADT1>& m1, const Mul<ADT2>& m2)
{ return eq(*m1.l_, *m2.l_) &&                        //= eq(l1,l2) ∧
    eq(*m1.r_, *m2.r_); }                               //eq(r1,r2)
template<typename ADT1, typename ADT2> bool
the_eq_match(const Sub<ADT1>& s1, const Sub<ADT2>& s2) {...}
template<typename ADT1, typename ADT2>        //eq(Neg(t1),Neg(t2))
bool the_eq_match(const Neg<ADT1>& g1, const Neg<ADT2>& g2)
{ return eq(*g1.t_, *g2.t_); }                          //= eq(t1,t2)
```

As will become clear in the subsequent sections, this is because, our solution is structural as opposed to nominal. That is, our dispatcher traverses the structure – i.e., the case list – of an ADT in order to find the right match component. Contrast that with the Scala IDPAM [22], in which the programmer needs to manually nominate the match components. In the C++ IDPAM, match components can also pertain to, say, all the binary operators, in which case they are even more structural. We do not present that here though.

The two-phase `template` translation [43, §14.3.1] of C++ is likely to cause confusion w.r.t. how C++ IDPAM manages to address EP's separate compilation concern. For a detailed discussion, see [18, §4.4].

2.2 Genericity

The reader may have noticed that the match components take two `template` parameters: `ADT1` and `ADT2`. That is on purpose: `eq` can structurally compare expressions from two different ADTs. Only the special case is when `ADT1` and `ADT2` are the same.

3 The One-Liner

Before the technical development of this section, we would like to explain a naming intention of ours: We chose the name "one-liner" because it takes only one line of code to **use** the macro `GENERATE_MULTIPLE_DISPATCH` for instructing the late-binding. As we are about to see in this section, definition of the one-liner, however, takes multiple dozens of lines.

It now is time to reveal the technology behind our one-liner. We begin in § 3.1 by presenting some utility macros used by the one-liner macro. In § 3.2, then,

we present how we generate the function to be multiply dispatched, a.k.a., the *call centre*. In § 3.3, we offer rewriting rules that formalise our C++ dispatcher. It is in § 3.4 that we detail the real dispatcher generated by the one-liner. We end this section in § 3.5 with deeper remarks about our C++ dispatcher.

In § 3.2, we frequently talk about introspection of an argument, an ADT case, or an ADT. The exact intention will come afterwards in § 3.3 and § 3.4.

3.1 Utilities

The two macros below facilitate name production for the different role players in the one-liner technology. They both do so by juxtaposition of tokens during macro expansion.

Given a token `name`, the macro `FUNC_MATCH(name)` expands to `"the_"` followed by `name` followed by `"_match"`. This is a naming convention in our codebase: For a function `f` on ADTs, the match components need to be called `the_f_match`. For example, note that with the second argument passed to the one-liner macro in § 2 being `eq`, the match components are called `the_eq_match`.

The macro `FUNC_DISPATCH(name)` expands similarly to `FUNC_MATCH(name)`. Again, this is a naming convention we follow: For a function `f` on ADTs, an internal set of **`template`** specialisations will be generated upon expansion of the one-liner macro that are all called `the_f_dispatcher`. Those **`template`** specialisations are at the core of the automation provided by the one-liner. See § 3.4 for more details.

3.2 Call Centre

Recall from § 2 that the one-liner for structural equality was called with `eq` as the second argument. Recall also that the user did not implement `eq` itself but used it. See line 22 of `the_eq_match` for an example in the match components and `eq(5_n, 5_n)` for concrete expression comparison.

The `eq` function is the contact point for the user of the multiple dispatch. Moreover, as will be seen below, all it does is to kick-start the recursive navigation of the relevant ADTs for pointer introspection. Therefore, we follow the C++ IDPAM in calling such functions the call centres. Our following preprocessor macro is for generating call centres:

```
1  #define GENERATE_CALL_CENTRE(return_type, function_name)      \
2  template<typename... Ts>                                      \
3  return_type function_name(const Ts&... xs) {                  \
4    static_assert(sizeof... (Ts) != 0, "nullary dispatch");     \
5    if constexpr(conjunction_v<is_adt<Ts>...>)                  \
6      return FUNC_DISPATCH(function_name)<                      \
7       tuple<>,                                                 \
8       tuple<Ts...>,                                            \
9       rendered_adt_cases_t<tuple_element_t<0, tuple<Ts...>>>   \
10     >::match(tuple<>(), xs...);                               \
11   else /* ... branches for other forms ... */                 \
12 }
```

The above macro takes two parameters: the name of the call centre (`function_name`) and its return type (`return_type`). It expands to a such-called function with the requested return type (i.e., the call centre) that is variadically parameterised by the types (`Ts...`) of its regular parameters (`xs...`).

The call centre first makes sure (in line 4) that it is called with at least one argument to be late-bound. Few call centre invocation forms are possible. We only show the code for processing one here: when all arguments are statically ATDs (first form). It is also possible that all arguments are statically of case component types (second form). Another form would be when the argument needs no late-binding, and, is, hence, of neither type. The final form is when different arguments are of different kinds above.

The first form is typically when the call centre is recursively invoked during structural decomposition of expressions. Recall, for example, that the data members `l_` and `r_` of `Add` are (shared) pointers to `ADT`. (C.f. line 8 of `Add`.) Consequently, the results of dereferencing `l_` and `r_` are of the static type `ADT`, as opposed to a case component's type. The call `eq(*a1.l_, *a2.l_)` in line 22 of `the_eq_match` for `Add` is an instance of the first form. The second form is when the call centre is invoked on concrete ADT expressions, as in `eq(5_n, 5_n)`, for instance.

We now explain the only branch of the call centre we show the code of, i.e., that for the first form.

When all the arguments are statically ADTs (line 5), the call centre kick-starts the dispatch process by invoking the `match` member function of the right `template` specialisation of the dispatcher. A full explanation about the dispatcher comes in § 3.4. Here, we only focus on explaining the current dispatcher invocation's type arguments (lines 7–9) and regular arguments (line 10).

The first type argument (line 7) is a tuple of the types of the arguments introspected thus far. Given that, at this point, no argument is still introspected, we are passing an empty tuple type (`tuple<>`) here. The second type argument is the ADTs not still fully introspected. That is currently all the ADTs, implying the pass of `tuple<Ts...>` in line 8. The third type argument is the type of the next ADT to be introspected. No ADT is introspected yet at this point. Hence, all the cases of the 0^{th} element in `tuple<Ts...>` are passed in line 9. (For more on `rendered_adt_cases_t`, see § A.)

The `match` member function takes two parameters. Those are, respectively, the arguments passed to the call centre that are thus far introspected and those that are still to be introspected. Given that, at this point, none of the arguments is still introspected, in line 10, an empty tuple (`tuple<>()`) is passed for the former, and the set of all arguments (`xs...`) is passed for the latter.

3.3 Rewriting Rules of The Dispatcher

In this section, we provide rewriting rules as a formal specification of our C++ dispatcher. We assume a best-fit strategy for applying the rules.

Those rules will also surmount the verbosity difficulty. Note that the verbosity of C++ metaprogramming can increase dramatically by complexity of code logic.

Indeed, the high syntactic noise in our dispatcher might make it impenetrable to the unfamiliar eye – hampering reusability of the technique. In § 3.4, we, hence, relate the metaprogramming to the corresponding rules for the idea to be clear when we navigate the syntax.

Our rewriting rules use the following list comprehension syntax: A list of elements $\overline{e}$ can either be empty (ϵ) or of the form $e_1 \dots e_n$. For an element e, the pieces of syntax $e.\overline{e}$ and $\overline{e}.e$ denote left and right concatenation, respectively. The rules themselves come below:

The Dispatcher function takes three type parameters: $\overline{\gamma}_a$, $\overline{\alpha}_x$, and $\overline{\gamma}_x$. It also takes two regular parameters: $\overline{a}$ and $\overline{x}$. Given that, in the metaprogramming, it is the call centre that first invokes the dispatcher, we rather refer to $\overline{a}$ and $\overline{x}$ as the regular *arguments* (passed to the call centre). As such, $\overline{a}$ are the arguments already introspected and $\overline{x}$ are those still to be introspected.

Here is the explanation of the three type parameters of the Dispatcher function: $\overline{\gamma}_a$ are the cases that $\overline{a}$ are instances of. $\overline{\alpha}_x$ are the ADTs of $\overline{x}$. And, $\overline{\gamma}_x$ are the remaining cases of $head(\overline{\alpha}_x)$, if any.

$$\frac{\gamma \in type(x) \qquad \overline{\alpha}'_x = \epsilon}{\text{Dispatcher}\langle\overline{\gamma}_a, \alpha.\overline{\alpha}'_x, \gamma.\overline{\gamma}'_x\rangle(\overline{a}, x.\overline{x}') \rightarrow \text{Dispatcher}\langle\overline{\gamma}_a.\gamma, \epsilon, \epsilon\rangle(\overline{a}.\dot{x}, \epsilon)} \; (S\#1-\gamma\epsilon)$$

$$\frac{\gamma \in type(x) \qquad \overline{\alpha}'_x \neq \epsilon}{\text{Dispatcher}\langle\overline{\gamma}_a, \alpha.\overline{\alpha}'_x, \gamma.\overline{\gamma}'_x\rangle(\overline{a}, x.\overline{x}') \rightarrow \text{Dispatcher}\langle\overline{\gamma}_a.\gamma, \overline{\alpha}'_x, cases(head(\overline{\alpha}'_x))\rangle(\overline{a}.\dot{x}, \overline{x}')} \; (S\#1-\gamma\not\epsilon)$$

$$\frac{\gamma \notin type(x) \qquad \overline{\gamma}'_x = \epsilon}{\text{Dispatcher}\langle\overline{\gamma}_a, \alpha.\overline{\alpha}'_x, \gamma.\overline{\gamma}'_x\rangle(\overline{a}, x.\overline{x}') \rightarrow \text{Dispatcher}\langle\overline{\gamma}_a, \overline{\alpha}_x, \epsilon\rangle(\overline{a}, \epsilon)} \; (S\#1-\not\gamma\epsilon)$$

$$\frac{\gamma \notin type(x) \quad \overline{\gamma}'_x \neq \epsilon \quad \overline{\alpha}_x = \alpha.\overline{\alpha}'_x \quad \overline{x} = x.\overline{x}'}{\text{Dispatcher}\langle\overline{\gamma}_a, \alpha.\overline{\alpha}'_x, \gamma.\overline{\gamma}'_x\rangle(\overline{a}, x.\overline{x}') \rightarrow \text{Dispatcher}\langle\overline{\gamma}_a, \overline{\alpha}_x, \overline{\gamma}'_x\rangle(\overline{a}, \overline{x})} \; (S\#1-\not\gamma\not\epsilon)$$

$$\frac{}{\text{Dispatcher}\langle\overline{\gamma}_a, \epsilon, \epsilon\rangle(\overline{a}, \epsilon) \rightarrow m\langle\overline{\gamma}_a\rangle(\overline{a})} \; (S\#2)$$

$$\frac{}{\text{Dispatcher}\langle\_,\_,\epsilon\rangle(\_,\_) \rightarrow \bot} \; (S\#3)$$

$$\frac{}{\text{Dispatcher}\langle\_,\_,\_\rangle(\_,\_) \rightarrow \bot} \; (S\#4)$$

Initially, all the (static) type information that is available about $\overline{x}$ is their ADTs. One after the other, arguments in $\overline{x}$ are tested, then, against each and every case in the case lists of their **own**[3] ADTs – unless the test succeeds. The

[3] See the closing paragraph of this section.

purpose of that test is to figure out whether the argument has also got the (dynamic) case type. (That is, whether the argument is an instance of the case.) Roughly speaking, the test goes on until it either succeeds or the case list of the argument's ADT runs out. In the former situation, the test moves to the next argument in $\overline{x}$, whilst an error is emitted in the latter.

What we referred to in the previous paragraph as *test*ing the $\overline{x}$ is a meaning for what we earlier called argument introspection, most notably in § 3.2. We will further elaborate on that when it comes to pointer introspection in § 3.4.

When the introspection of an argument x is done, its case type is bookmarked for future in $\overline{\gamma}_a$. Additionally, x itself dynamically cast into its case type – denoted in the rewriting rules by $\dot{x}$ – is also stored in $\overline{a}$. Accordingly, once all the arguments are introspected, all the case types are known and the match component of the right type ($\overline{\gamma}_a$) can be called on correctly cast arguments ($\overline{a}$).

The rewriting rules proceed by case distinction on $\overline{\alpha}_x$ and $\overline{\gamma}_x$. Four scenarios are possible: $S\#1$ to $S\#4$. ($S\#1$ further divides into sub-scenarios.)

The ultimate goal is to reach to $S\#2$, where the introspection has succeeded for all the arguments and the right match component can be called using the accumulated type information on the right-typed arguments ($\overline{a}$). (That match component is denoted by $m{<}\overline{\gamma}_a{>}$ in ($S\#2$).) Conceptually, two erroneous scenarios are also possible. $S\#3$ is the scenario where the case list of an argument's ADT has run out without success. $S\#4$ is when some unexamined cases still remain but the list of ADTs has run out.

$S\#1$ is less trivial: the scenario when there are still both more arguments to introspect and more cases in the pertaining ADT's case list. It is either, then, that the first argument (x) is an instance of the first case (γ) or not. We denote that in the rewriting rules by $\gamma \in \mathit{type}(x)$.

If $\gamma \in \mathit{type}(x)$, two possibilities exist: When the current argument was the very last for introspection ($S\#1-\gamma\epsilon$), we bookmark the respective case type (γ) and the type-tuned argument ($\dot{x}$), and, recursively call Dispatcher with a signal for its task coming to an end. (The signal is emptiness of both type parameters $\overline{\gamma}_a$ and $\overline{\alpha}_x$.) When more arguments are to be introspected ($S\#1-\gamma\not\epsilon$), we recursively call Dispatcher by similar bookmarking to start introspection of the next argument against cases of the next ADT (i.e., $\mathit{head}(\overline{\alpha}'_x)$).

If $\gamma \notin \mathit{type}(x)$, again, two possibilities exist: When the cases have run out ($S\#1-\not\gamma\epsilon$), we recursively call Dispatcher with a signal for error. (The signal is passing a non-empty $\overline{\alpha}_x$ – to indicate remaining arguments to be introspected – and empty $\overline{\gamma}_x$ – to indicate run out of the case list.) Otherwise ($S\#1-\not\gamma\not\epsilon$), we recursively call Dispatcher to introspect the same argument (x) using the remaining cases in the same case list ($\overline{\gamma}'_x$).

One may wonder how we make sure that every x is paired in Dispatcher by its **own** ADT α. That can easily be proved by rule-based induction. For the basis of induction, note that `const Ts&...` in the call centre are the types of `xs...`; for the inductive step, note that all the rules preserve that relationship between $\overline{x}$ and $\overline{\alpha}$.

3.4 The C++ Dispatcher

A distinctive difference between the rewriting rules and the C++ dispatcher is that type parameters of the latter are type **tuples**. This is because, as will be seen below, the C++ dispatcher employs variadic `template`s. According to the C++ restrictions for variadic `template`s, variadic type parameters can only be positioned at the end of the type parameter list. Hence, if a `template` specialisation needs to employ variadic type parameters at other positions, those type parameters are typically bundled inside `tuple`s. In that vein, the C++ dispatcher takes `tuple` type parameters that represent the type parameters of the term rewriting.

We now walk the reader through the definition of the one-liner macro:

```
1  #define GENERATE_MULTIPLE_DISPATCH(return_type,function_name)\
2  template<                                                    \
3    typename GammaAsTup,                                       \
4    typename AlphaXsTup,                                       \
5    typename GammaXsTup                                        \
6  > struct FUNC_DISPATCH(function_name);                       \
```

We keep the names of the C++ type parameters as close to their term rewriting counterparts as possible. For example, in lines 3–5 above, `GammaAsTup`, `AlphaXsTup`, and `GammaXsTup` are the `tuple`s for $\overline{\gamma}_a$, $\overline{\alpha}_x$, and $\overline{\gamma}_x$, respectively.

Lines 2–6 above produce the general `template` signature for the C++ dispatcher. Four `template` specialisations, then, implement the scenarios $S\#1$ to $S\#4$ in the rewriting rules.

The patterns we match type arguments against in the four scenarios – $S\#3$, in particular – imply using partial `template` specialisation. But, C++ does not offer that for functions at the moment – leaving us to `class`es or `struct`s. We choose the latter for there is no need for encapsulation for the four scenarios. Those scenarios, however, need to also have call signatures. For each `template` specialisation, therefore, we provide a `static` member function that is called `match`, by convention. Those `match` member functions get called recursively according to the rewriting rules.

We now present those `template` specialisations one after the other.

Scenario S#1 The implementation of $S\#1$ starts by pattern matching on the $\overline{\alpha}_x$ and $\overline{\gamma}_x$. Those are lines 15 and 16 below, respectively.

```
7  template                                                     \
8  <                                                            \
9    typename... GammaAs, typename Alpha, typename... AlphaXsPr,\
10   typename Gamma, typename... GammaXsPr                      \
11 >                                                            \
12 struct FUNC_DISPATCH(function_name)                          \
13 <                                                            \
14   tuple<GammaAs...>,                                         \
15   tuple<Alpha, AlphaXsPr...>,                  /* ᾱ_x = α.ᾱ'_x */\
```

```
16    tuple<Gamma, GammaXsPr...>                          /* γ̄_x = γ.γ̄'_x */\
17  >                                                     /*  S#1          */\
```

The `match` member function takes three regular parameters (lines 19 and 21 below): `as_tup` for the `tuple` of $\overline{a}$, `x` for x, and `xs_pr` for $\overline{x}'$.

```
18  {                                                                       \
19    static return_type match(const tuple<GammaAs...>& as_tup,            \
20                             const Alpha& x,                              \
21                             const AlphaXsPr&... xs_pr)                   \
22    {                                                                     \
23      const auto cp = dynamic_cast<const Gamma*>(&x);                     \
24      if(cp) {                                                            \
```

Note that the static type of `x` is known only to be (reference to constant) `Alpha` – i.e., its ADT. In order to figure out the case which `x` is an instance of, similar to $\gamma \in type(x)$ in the term rewriting, we check `x`'s dynamic type. Thus, lines 23–24 above are our promised pointer introspection. In line 23, we are storing the **`dynamic_cast`**'s result (line 23) in `cp` for later use. In line 24, we check if the pointer is not null, implying success of the **`dynamic_cast`**.

```
25        if constexpr (sizeof... (AlphaXsPr) == 0)    /* ᾱ'_x = ϵ */\
26          return apply(FUNC_DISPATCH(function_name)                   \
27                       <                                              \
28                         tuple<GammaAs..., Gamma>,     /* γ̄_a.γ */\
29                         tuple<>, tuple<>              /*  ϵ, ϵ  */\
30                       >::match,                                      \
31                      tuple_cat(as_tup,      /* ā, (*cp) ≅ ẋ */\
32                                make_tuple(cref(*cp))));             \
```

The mathematical comments above that relate to our term rewriting are expected to make following the logic easy. Here is explanation on the remaining C++ peculiarities: `tuple_cat` takes two `tuple`s and return their concatenation. Our usage of `cref` is to guide automatic type deduction. The `make_tuple` in line 32 builds a singleton `tuple` out of an element. The need for using `apply` will become clear in lines 65 and 66 below.

```
33        else                                                 /* ᾱ'_x ≠ ϵ */\
34          return FUNC_DISPATCH(function_name)                         \
35                 <                                                    \
36                   tuple<GammaAs..., Gamma>,          /* γ̄_a.γ */\
37                   tuple<AlphaXsPr...>,               /*  ᾱ'_x  */\
38                   rendered_adt_cases_t     /* cases of head(ᾱ'_x) */\
39                     <tuple_element_t<0, tuple<AlphaXsPr>>>     \
40                 >::match(tuple_cat(as_tup,                /* ā */\
41                                    make_tuple(cref(*cp))),/*ẋ*/\
42                          xs_pr...);                       /*  x̄'  */\
43      }/* if(cp) */                                                   \
```

Most of the C++ syntax for the above **`else`** branch (for when x is an instance of γ but $\overline{\alpha}'_x \neq \epsilon$) is similar to the corresponding **`if`** branch. `rendered_adt_cases_t` type evaluates to the case list of its type argument.

See § A for more details. `tuple_element_t` is metafunction taking two type parameters: an index i and a `tuple` type. It type evaluates to the i^{th} type element in its second argument, if any. (Otherwise, the compilation fails.)

Below comes the code for when x is not an instance of γ. We believe the inlined mathematical comments in addition to C++ syntax formerly explained in this section should make the code readable. We drop further explanation thus.

```
44      else {                                              /* γ'_x = ϵ */\
45        if constexpr(sizeof... (GammaXsPr) == 0)                       \
46          return FUNC_DISPATCH(function_name)                          \
47                 <                                                     \
48                   tuple<GammaAs...>,                      /* γ_a */\
49                   tuple<Alpha, AlphaXsPr...>,             /*α.ᾱ'_x*/\
50                   tuple<>                                 /* ϵ */\
51                 >::match(as_tup);                                     \
52        else                                               /* γ'_x ≠ ϵ */\
53          return FUNC_DISPATCH(function_name)                          \
54                 <                                                     \
55                   tuple<GammaAs...>,                    /* γ_a */\
56                   tuple<Alpha, AlphaXsPr...>,           /* α.ᾱ'_x */\
57                   tuple<GammaXsPr...>                   /* γ'_x */\
58                 >::match(as_tup, x, xs_pr...);          /*ā,x,x̄'*/\
59      }                                                              \
60    }/* match */                                                    \
61  };                          /* S#1's template specialisation */\
```

The Rest We now move to the remaining lines of our one-liner's definition.

```
62  template<typename... GammaAs>                                      \
63  struct FUNC_DISPATCH(function_name)                  /* S#2 */\
64    <tuple<GammaAs...>, tuple<>, tuple<>> {            /* γ,ϵ,ϵ */\
65    static return_type match(GammaAs... as)                          \
66    { return FUNC_MATCH(function_name)(as...); }                     \
67  };                                                                 \
```

The crucial difference between the above specialisation and the previous one is in the `match` member function. The one in line 65 accepts variadic arguments, as opposed to a `tuple` of variadic size. This is because of line 66 where it passes the same arguments to the right match component. Recall from § 2 that, for usage simplicity, the match components accept such arguments too.

We can now relate back to line 26. `apply` takes a callable and a `tuple` and invokes its first argument against the contents of the second. Recall that, in line 26, $\overline{a}.\dot{x}$ is stored as a `tuple`. Hence, `apply` is required for passing it to the `match` member function in line 65. Note that automatic type deduction fails if one rather passes the `tuple` in line 26 and employs `apply` in line 66. The compiler would have no way to figure out which overload of the match components it should call with `apply`. In lines 28–29, however, we manually inject the right types into the type system. Such a manual type-injection is not possible for

`function_name` itself. That is because, according to the C++ parsing rules, `function_name<GammaAs..., Gamma>` does not constitute a well-formed token.

Lines 65 and 66 are of key importance in our C++ codebase. This is because, when the execution reaches there, it has finally made it to the specialisation where the case types are all known **statically**. (This can be after many recursive calls to the `match` member functions of other specialisations. See § 3.5 for more.) It is only then that overload resolution – which is done statically and according to argument types – can help the choice of the right match component.

```
68 template<typename... GammaAs, typename AlphaXsTup>                \
69 struct FUNC_DISPATCH(function_name)                  /* S#3 */\
70   <tuple<GammaAs...>, AlphaXsTup, tuple<>> {      /* γ_a,_,ϵ */\
71   static return_type match(const tuple<GammaAs...>& as_tup)  \
72   { throw logic_error                                        \
73       ("argument not an instance of its ADT's case list"); } \
74 };                                                           \
```

What is noteworthy about the above implementation of $S\#3$ is it needs not to be explicit about the constraint $\overline{\alpha}_x \neq \epsilon$ on its second type argument. This is because, according to C++ rules, $S\#2$'s implementation will automatically be chosen when $\overline{\alpha}_x = \epsilon$ because $S\#2$'s `template` specialisation is more specific then. $S\#3$ is the most specific specialisation otherwise. The situation is similar for the third type parameter of $S\#4$'s `template` specialisation below.

```
75 template<typename... GammaAs, typename GammaXsTup>                \
76 struct FUNC_DISPATCH(function_name)                  /* S#4 */\
77   <tuple<GammaAs...>, tuple<>, GammaXsTup> {      /* γ_a,ϵ,_ */\
78   static return_type match(const tuple<GammaAs...>& as_tup)  \
79   { throw logic_error                                        \
80     ("case list for a non-existing ADT"); }                  \
81 };                                                           \
82                                                              \
83 GENERATE_CALL_CENTRE(return_type, function_name)
```

Et voilà! That is our one-liner macro defined in less than 100 lines.

3.5 Rumination

Combinatorial Explosion. The reader might have noticed the combinatorial explosion of `template`s instantiated by our multiple dispatch technology. Indeed, the number of `template`s that might be instantiated over compilation of a call centre invocation is $\mathcal{O}(a \times \alpha \times m)$, where a is the invocation's arity, α is the number of ADTs in the arguments, and $m = \max\{\#(cases(\alpha)) \mid \alpha \in \overline{\alpha}\}$. This is the case even when the expressions do not use all the cases of every ADT's case list.

Nevertheless, we use `struct`s for scenarios $S\#1$ to $S\#4$, hinting at simplicity of their bodies to the compiler for inlining. By making the `match` member functions `inline`, one can also give the compiler a hint about replacing calls to `match` directly with their bodies. Those kinds of optimisation are likely but not guaranteed. See [43, §23.3] for more.

Ambiguity and Unavailability. Two important factors for measuring performance of a solution to multiple dispatch are ambiguity of dispatch and unavailability of a match. The former is when, given the arguments, more than one match is available for the dispatch. The latter is the lack of any match for the passed arguments.

Match statements in our solution are simple function overloads. Consequently, both ambiguity and unavailability cause failure in overload resolution amongst match component, and, hence, compile-errors. That is, when it comes to ambiguity or unavailability, the programmer is in charge of disambiguation or provision of what is missing. We chose not to make a decision on their behalf because C++ programmers are already skilful at dealing with overload resolution.

4 Case Studies

4.1 Higher Arity

In this case study, we tested our technology against multiple dispatch on functions of higher arity. For example, consider the following ADTs: $\mathit{Personel} = \mathit{Boss} \oplus \mathit{Employee} \oplus \mathit{Guest}$, $\mathit{Priority} = \mathit{Urgent} \oplus \mathit{High} \oplus \mathit{Normal} \oplus \mathit{Low}$, and $\mathit{Document} = \mathit{Article} \oplus \mathit{Book}$. Multiple dispatch for the following function *rank* is as easy as `eq` in § 2. $\mathit{rank} : \mathit{Personel} \times \mathit{Priority} \times \mathit{Document} \to \mathbb{N}$ gives a rank to a printing task based on: the position of the task owner in the employment hierarchy, the urgency associated to the task, and the document type.

4.2 Core Lazy Calculi

In this case study, we embedded a family of eleven core lazy calculi[4] in C++. Multiple dispatch is a key ingredient for implementing formal semantics. Our technology easily accomplished that. We now summarise the highlights of the experiment, albeit, using the running example of this paper.

Inherited from the C++ IDPAM, pointers are inevitable for accommodating recursive nature of ADTs. Hence, the programmer has to be skilled in pointer arithmetics. Misplacing the dereference operator (say writing `(*a1).l_` instead of `*a1.l_` in line 22) can lead to cryptic error novels. One needs to also know exactly where to put `msfd`. Otherwise, the iterative pointer introspection will run out, causing a runtime exception. (See Appendix A for more on `msfd`.)

Storing the results of computations in `auto` variables can cause surprise. Runtime polymorphism may be required for the return value of functions, in which case, a `shared_ptr` needs to be returned. Consider a function `shared_ptr<ADT > abs(const ADT&)`, for example. Attempting `auto abs_x = *abs(x)` may, then, not work as expected. The reason is that a C++ compiler is free to choose its convenient return type for the dereference `operator` of a `shared_ptr`. Whilst

[4] Abramsky and Ong [1], Launchbury [26], van Eekelen and de Mol [42], Sinot [37], Haeri [16][17, §8.4.3], Haeri and Shupp [20], Jost et al. [25], Bischof et al. [7], Hacket and Hutton [15], and Hanus [23,24]

the result needs to be implicitly convertible to the underlying type, the stored type is not necessarily the underlying type itself. Accordingly, chaining of function calls works perfectly (say as in `to_string(*abs(x))`. But, for storing return values, more specific type annotation is needed at variable definition site.

Further generalised one-liners are required when dealing with functions with return types that depend on the type parameters of the call centre. As an example, consider the function `abs` in the previous paragraph, which returns a `shared_ptr<ADT>`. In some such cases, automatic type deduction will not be possible for the compiler, dictating explicit type guidance. Luckily, one can always wrap the real multiply dispatched function inside another, inside which the type calculation for guiding the compiler is performed according to the argument types—retaining automatic type deduction for the user. For example, suppose that obtaining the abstract value requires the explicit type guidance. One can, then, have an `abs_helper` multiply dispatched instead, around which `abs` wraps (with automatic type deduction).

5 Related Work

For extensive surveys on multiple dispatch, one can see [13] and [32]. In this section, we study related works in terms of their extensibility and handling of ambiguity and unavailability. Recall that our technology is extensible (§ 2.1) and detects ambiguity and unavailability statically (§ 3.5).

5.1 Libraries

Alexandrescu [2, §11] was the first to use mechanical type list traversal for pointer introspection to solve multiple dispatch in C++. Yet, in his solutions, explicitly nominated call-backs are used for the match statements. As such, when new cases are added to an ADT, his dispatcher needs to be reformulated to accommodate the new call-backs. Compilation fails in the presence of ambiguity or unavailability.

Smith proposes Cmm [38] for making use of preprocessors for an open-method C++ extension. However, Cmm is off-line in that it processes C++ translation units to generates their dispatch code – implying loss of extensibility. Cmm throws runtime exceptions for ambiguity and unavailability.

Bettini, Capecchi, and Venneri offer DoubleCpp [4] for converting code with double dispatch to the equivalent that uses the famous visitor design pattern [14]. DoubleCpp automatically decides on resolving an ambiguity without giving the programmer the possibility for intervention. Unavailability will cause a linkage failure in DoubleCpp. Extensibility of DoubleCpp is unclear to us.

Solodkyy, Dos Reis, and Stroustrup [39] offer *Mach7* as a library on top of the Concepts Lite [41] extension to C++. The extension is akin to the pattern matching of functional programming, which, obviously supports multiple dispatch too. The difference is that ambiguity and unavailability cause compile-errors. *Mach7* allows hierarchical extensibility.

Leroy's Yomm2 [27] is a library embodiment of the open methods model of Pirkelbauer, Solodkyy, and Stroustrup [33,34]. Instead of using the `virtual` keyword of C++ as prescribed by the latter works, Yomm2 provides the `virtual_` metafunction to guide its multiple dispatch engine. Like our machinery, Yomm2 relies heavily on metaprogramming. Yet, its macros are required at far more places than us. Besides, the resulting code, after usage of those macros, may look eccentric in comparison to mainstream C++. Yomm2 reports ambiguity at runtime and fails to link in the presence of unavailability. Yomm2 requires recompilation upon extension of ADTs but not provision of new functions on them.

5.2 Informal Models

Wonnacott [45] has one concern out of the four EP concerns for enabling multiple dispatch. Yet, his work is not an EP solution in that it proceeds by recommending a change to the common virtual table technologies of the time.

Pirkelbauer, Solodkyy, and Stroustrup [33,34] propose open methods as an extension to C++98. Their proposal is for annotation of virtual parameters with the `virtual` keyword of C++. Our match components too are open methods in that they do not ship as a member functions of the receiver classes. Open methods resolve ambiguity based on covariant return types and fails to compile for unavailability.

Bettini, Capecchi, and Venneri propose FDJ [5] for dynamic overload resolution with translation to alleged equivalents using familiar function overloading and single dispatch. They detect ambiguity and unavailability statically. Addition of new ADTs, ADT cases, or new functions defined on ADTs entails recompilation in FDJ.

Demaille [12] proposes a C++ model for runtime `template` instantiation. Amongst other things, his proposal endorses multiple dispatch in a fashion that new match statements can be loaded at runtime.

Clifton et al. [10] propose a Java model for multiple dispatch that supports only limited extensibility.

5.3 Calculi

Lievens and Harrison [28,29] propose a calculus for multiple dispatch that accommodates modular extensibility. In their calculus, multi-methods can be defined in either receiver class – catering robustness against change. Furthermore, functions defined in one class may be overloaded by functions defined in classes other than subclasses. They prove the soundness of their results.

Malayeri and Aldrich [30] propose a calculus for multiple dispatch that prevents ambiguity in the match statements by prohibiting diamond inheritance, whilst still allowing multiple inheritance. They prove that their calculus manages modular type-checking of multi-methods without hampering expressiveness.

In a third work of theirs [6], Bettini, Capecchi, and Venneri offer a calculus for dynamic overload resolution that can statically detect dispatch ambiguities.

6 Conclusion

In this paper, we present a solution to the classical problem of multiple dispatch. Our solution orchestrates an emulation of the familiar pattern matching of functional programming. Using our solution is as easy as specifying the match statements in the form of function overloads and calling a one-liner preprocessor macro. Behind the scenes, in less than 100 lines of code, the one-liner generates few `template` specialisations. The duty of those `template` specialisations is to mechanically traverse the cases of the respective ADTs for pointer introspection until a match is found, if any.

Achieving that solution in C++ is possible because of C++'s `tuple` and variadic pattern matching on that. We use `tuple` for compile-time specification of cases of ADTs. Pattern matching on `tuple` is what we use for mechanical traversal of the cases of ADTs for pointer introspection at runtime. Nevertheless, our technique is applicable in any language in which the case list of an ADT can be programmatically traversed for introspection.

For the future work, we plan to test scalability of our technology. That we do via embedding larger-scale languages in C++. A medium-scale choice would be CinK of Mosses and Vesely [31]. Larger scale choices we have in mind are the component-based semantics of CAML LIGHT [9] and the modular implementation of OBERON-0 [3].

References

1. S. Abramsky and C.-H. Ong. Full Abstraction in the Lazy Lambda Calculus. *Inf. & Comp.*, 105(2):159–267, August 1993.
2. A. Alexandrescu. *Modern C++ Design: Generic Prog. & Design Patterns Applied.* AW Longman Pub., Boston, MA, USA, 2001.
3. B. Basten, J. van den Bos, M. Hills, P. Klint, A. Lankamp, B. Lisser, A. van der Ploeg, T. van der Storm, and J. J. Vinju. Modular Language Implementation in Rascal — Experience Report. *SCP*, 114:7–19, 2015.
4. L. Bettini, S. Capecchi, and B. Venneri. Double Dispatch in C++. *J. SPE*, 36(6):581–613, 2006.
5. L. Bettini, S. Capecchi, and B. Venneri. A Safe Implementation of Dynamic Overloading in Java-Like Languages. In F. Arbab and M. Sirjani, editors, 3^{rd} *FSEN Revised Selected Papers*, volume 5961 of *LNCS*, pages 455–462. Springer, April 2009.
6. L. Bettini, S. Capecchi, and B. Venneri. Featherweight Java with Dynamic and Static Overloading. *SCP*, 74(5-6):261–278, 2009.
7. S. Bischof, J. Breitner, D. Lohner, and G. Snelting. Illi Isabellistes Se Custodes Egregios Praestabant. In P. Müller and I. Schaefer, editors, *Prin. Soft. Dev. – Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his* 60^{th} *Birthday*, pages 267–282. Springer, 2018.
8. P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-Bounded Polymorphism for Object-Oriented Programming. In 4^{th} *FPCA*, pages 273–280, September 1989.
9. M. Churchill, P. D. Mosses, N. Sculthorpe, and P. Torrini. Reusable Components of Semantic Specifications. *TAOSD*, 12:132–179, 2015.

10. C. Clifton, G. T. Leavens, C. Chambers, and T. D. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In 15^{th} *OOPSLA*, pages 130–145, Minneapolis, Minnesota, USA, 2000. ACM.
11. W. R. Cook. Object-Oriented Programming Versus Abstract Data Types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *FOOL*, volume 489 of *LNCS*, pages 151–178, Noordwijkerhout (Holland), June 1990.
12. A. Demaille. Runtime Template Instantiation for C++. *CoRR*, abs/1611.00947, 2016.
13. K. Driesen. Multiple Dispatch Techniques: A Survey, August 2005. Available on CiteSeerX under the doi 10.1.1.38.8841.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. AW Professional, October 1994.
15. J. Hackett and G. Hutton. Call-By-Need is Clairvoyant Call-By-Value. 2019.
16. S. H. Haeri. Observational Equivalence and a New Operational Semantics for Lazy Evaluation with Selective Strictness. In *TMFCS-10*, pages 143–150, 2010.
17. S. H. Haeri. *Component-Based Mechanisation of Programming Languages in Embedded Settings*. PhD thesis, STS, TUHH, Germany, December 2014.
18. S. H. Haeri and P. W. Keir. Metaprograms to Help Expressions with their Problem. available online at: https://archive.org/details/ictac2019, September 2019.
19. S. H. Haeri and S. Schupp. Reusable Components for Lightweight Mechanisation of Programming Languages. In W. Binder, E. Bodden, and W. Löwe, editors, 12^{th} *SC*, volume 8088 of *LNCS*, pages 1–16. Springer, June 2013.
20. S. H. Haeri and S. Schupp. Distributed Lazy Evaluation: A Big-Step Mechanised Semantics. In *4PAD'14*, pages 751–755. IEEE, February 2014.
21. S. H. Haeri and S. Schupp. Expression Compatibility Problem. In J. H. Davenport and F. Ghourabi, editors, 7^{th} *SCSS*, volume 39 of *EPiC Comp.*, pages 55–67. EasyChair, March 2016.
22. S. H. Haeri and S. Schupp. Integration of a Decentralised Pattern Matching: Venue for a New Paradigm Intermarriage. In M. Mosbah and M. Rusinowitch, editors, 8^{th} *SCSS*, volume 45 of *EPiC Comp.*, pages 16–28. EasyChair, April 2017.
23. M. Hanus. Combining Static and Dynamic Contract Checking for Curry. In F. Fioravanti and Gallagher J. P., editors, 27*th LOPSTR Revised Selected Papers*, volume 10855 of *LNCS*, pages 323–340. Springer, October 2018.
24. M. Hanus. Improving Residuation in Declarative Programs. In J. J. Alferes and M. Johansson, editors, 21^{st} *PADL*, volume 11372 of *LNCS*, pages 82–97. Springer, January 2019.
25. S. Jost, P. Vasconcelos, M. Florido, and K. Hammond. Type-Based Cost Analysis for Lazy Functional Languages. *J. Auto. Reason.*, 59(1):87–120, 2017.
26. J. Launchbury. A Natural Semantics for Lazy Evaluation. In 20^{th} *POPL*, pages 144–154. ACM, 1993.
27. J.-L. Leroy. Yomm2, 2013. Online at Leroy's Multi-methods 1.0 website.
28. D. Lievens and W. Harrison. Symmetric Encapsulated Multi-Methods to Abstract over Application Structure. In S. Y. Shin and S. Ossowski, editors, 24^{th} *SAC*, pages 1873–1880. ACM, March 2009.
29. D. Lievens and W. Harrison. Abstraction over Implementation Structure with Symmetrically Encapsulated Multimethods. *SCP*, 78(7):953–968, 2013.
30. D. Malayeri and J. Aldrich. CZ: Multimethods and Multiple Inheritance without Diamonds. Technical report, CMU, School of Comp. Sci., December 2009. CMU-CS-09-153.

31. P. D. Mosses and F. Vesely. FunKons: Component-Based Semantics in $\mathbb{K}$. In S. Escobar, editor, *W. Rewrit. Logic & App.*, volume 8663 of *LNCS*. Springer, April 2014.
32. R. Muschevici, A. Potanin, E. D. Tempero, and J. Noble. Multiple Dispatch in Practice. In G. E. Harris, editor, 23^{rd} *OOPSLA*, pages 563–582. ACM, 2008.
33. P. Pirkelbauer, Y. Solodkyy, and B. Stroustrup. Open Multi-Methods for C++. In C. Consel and J. L. Lawall, editors, 6^{th} *GPCE*, pages 123–134. ACM, October 2007.
34. P Pirkelbauer, Y. Solodkyy, and B. Stroustrup. Design and Evaluation of C++ Open Multi-Methods. *SCP*, 75(7):638 – 667, 2010.
35. R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 7^{th} edition, 2009.
36. J. C. Reynolds. User-Defined Types and Procedural Data Structures as Complementary Approaches to Type Abstraction. In S. A. Schuman, editor, *New Direc. Algo. Lang.*, pages 157–168. INRIA, 1975.
37. F.-R. Sinot. Complete Laziness: a Natural Semantics. *ENTCS*, 204:129–145, April 2008.
38. J. Smith. Draft proposal for adding multimethods to C++. Technical Report N1463, JTC1/SC22/WG21 C++ Standards Committee, 2003.
39. Y. Solodkyy, G. Dos Reis, and B. Stroustrup. Open Pattern Matching for C++. In J. Järvi and C. Kästner, editors, *GPCE'13*, pages 33–42. ACM, October 2013.
40. I. Sommerville. *Software Engineering*. AW Pub., 9^{th} edition, 2011.
41. A. Sutton, B. Stroustrup, and G. Dos Reis. Concepts Lite: Constraining Templates with Predicates. Technical report, C++ Standards Committee, 2013. WG21/N3580, JTC1/SC22/WG21.
42. M. van Eekelen and M. de Mol. *Reflections on Type Theory, λ-Calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60^{th} Birthday*, chapter Proving Lazy Folklore with Mixed Lazy/Strict Semantics, pages 87–101. Radboud U. Nijmegen, 2007.
43. D. Vandevoorde, N. M. Josuttis, and D. Gregor. *C++ Templates: The Complete Guide*. Addison Wesley, 2^{nd} edition, 2017.
44. P. Wadler. The Expression Problem. Java Genericity Mailing List, November 1998.
45. D. Wonnacott. Using Accessory Functions to Generalize Dynamic Dispatch in Single-Dispatch Object-Oriented Languages. In R. Raj and Y.-M. Wang, editors, 6^{th} *COOTS*, pages 93–102. USENIX, February 2001.

A C++ IDPaM Technicality

The name "_" is a simple wildcard in C++ IDPaM: "`struct _ {};`".

As detailed in § 3.4, our pointer introspection is by `dynamic_cast`ing (shared) pointers to ADTs to those of ADT cases. Recall from § 2 that ADT cases derive from their respective ADTs, forming a hierarchy. C++, however, only allows `dynamic_cast` in the potential of virtual dispatch for a hierarchy. That is when the hierarchy has at least one `virtual` member function. That is why ADTs derive from the (dummy) class `Namer` below.

```
struct Namer { virtual string name() const {return "";} };
```

The C++ IDPAM ships with its own coding conventions. Here is an example that we observe in this paper: For an ADT $\alpha = \gamma_1 \oplus \gamma_2 \oplus \cdots \oplus \gamma_n$, the C++ specification is done by

```
1  template<> struct adt_cases<Alpha>
2  { using type = tuple<Gamma_1<>, Gamma_2<>, ..., Gamma_n<>>; };
```

(Note that the "..." in line 2 above is not for a `template` parameter pack. That is a meta-syntactic sugar.)

`adt_cases` is a unary metafunction. The above definition assumes the existence of case components `Gamma_1`, `Gamma_2`, ..., `Gamma_n`. Note that, for convenience, passing no type argument is required for case components in line 2 above. And, in fact, only those type arguments registered as ADTs will not be statically rejected. Nevertheless, replacing, say `Gamma_1<>` above by `Gamma_1<Alpha_pr>` (where `Alpha_pr` is another ADT) will not change anything. This is because, using the case list of an ADT is via `rendered_adt_cases_t`, which renders the right case list.

Note that `adt_cases<Alpha>::type` type evaluates to `tuple<Gamma_1<_>, Gamma_2<_>, ..., Gamma_n<_>>`. However, `rendered_adt_cases_t<Alpha>` type evaluates to `tuple<Gamma_1<Alpha>, Gamma_2<Alpha>,..., Gamma_n<Alpha>>`. The first element in the former would have changed to `Gamma_1<Alpha_pr>`, had we replaced `Gamma_1<>` in line 2 by `Gamma_1<Alpha_pr>`. The latter would have remained intact.

`msfd` (for *make shared from derived*) is a function that takes by `const`-reference an instance of a case and returns a `shared_ptr` pointing to a copy of it. The type argument of the `shared_ptr` is the ADT, whilst the dynamic type of its pointee is that of a case type.

```
1  template<typename, typename> struct MSFD;
2  template<typename B, typename D, typename... Ds>
3  struct MSFD<B, tuple<D, Ds...>> {
4    static shared_ptr<B> match(const B& x) {
5      const auto p = dynamic_cast<const D*>(&x);
6      return p? make_shared<D>(*p):
7                MSFD<B, tuple<Ds...>>::match(x);
8    }
9  };
10 template<typename B> struct MSFD<B, tuple<>> {
11   static shared_ptr<B> match(const B& x)
12   { return make_shared<B>(x); }
13 };
14 template<typename ADT>
15 inline shared_ptr<ADT> msfd(const ADT& x) {
16   return MSFD<
17               ADT, typename rendered_adt_cases<ADT>::type
18              >::match(x);
19 }
```

In line 6 above, a pointer introspection similar to that in line 23 of $S\#1$'s `template` specialisation takes place.